AFRL-IF-RS-TR-2003-254
Final Technical Report
October 2003

AN ASPECT-ORIENTED SECURITY ASSURANCE SOLUTION

Cigital Labs

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J756

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-254 has been reviewed and is approved for publication.

APPROVED: /s/
NANCY A. ROBERTS
Project Engineer

FOR THE DIRECTOR: /s/
JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	OCTOBER 2003	Final Apr 00 – May 03

4. TITLE AND SUBTITLE
AN ASPECT-ORIENTED SECURITY ASSURANCE SOLUTION

5. FUNDING NUMBERS
C - F30602-00-C-0079
PE - 63760E
PR - IAST
TA - 00
WU - 12

6. AUTHOR(S)
Viren Shah

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Cigital Labs
21351 Ridgetop Circle
Dulles Virginia 20166

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFTB
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2003-254

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Nancy A. Roberts/IFTB/(315) 330-3566/ Nancy.Roberts@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

Current tools and techniques to address security issues in a structured, comprehensive manner are sadly lacking. The most common method of addressing security flaws in software systems is the “penetrate-and-patch” approach. This project examined the viability of using the aspect-oriented programming paradigm to address security issues. A flexible framework based on this paradigm, and several security aspects were developed demonstrating this approach. Aspects developed ranged from ones that addressed the most common causes of security exploits, such as buffer overruns, race conditions and format strings, to higher-level and more complex issues such as type safety and event ordering.

14. SUBJECT TERMS
Aspect Oriented Programming, Security Aspects, Aspect Oriented Language, Information Assurance, Security Framework

15. NUMBER OF PAGES
57

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

LIST OF FIGURES

1. Executive Summary

In this project, we examined the viability of using the aspect-oriented programming paradigm to address security issues. Our research focused on researching and developing a flexible framework, based on aspect-oriented programming precepts, that was focused solely on addressing security issues. This project was unique in that no previous effort has applied the aspect-oriented programming concepts towards security. The goal was to advance the state-of-the-art in automated security solutions. The tasks for the project were divided into three high-level threads.

The primary thread was researching and devising a language that could be used to write the security aspects. This is of critical importance to the success of the project, since the capabilities of the language would determine the flexibility and usefulness of the security framework. The gist of the task involved defining the solution space that the framework would address, and then tailoring an aspect language towards it. Another important part of this task was to ensure that the language was simple enough so as to not present a steep learning curve for users. Over the course of the project, we researched, devised and refined our aspect language such that it is now simple yet flexible and can be used to address a wide variety of security issues.

The second thread, that took place in parallel with the language thread, was that of designing and building the infrastructure for the security framework. The main task here was to build an aspect weaver that would be the central transformation engine for the framework. The aspect weaver is the component that enables and performs the code transformations that are encoded in the aspects that are written using the aspect language. We built an aspect weaver that performed according to the required specifications. As a part of this component, we also developed a novel in-memory representation for code that enables code transformations easily.

The third thread, and the one that constituted the largest effort, was the creation of appropriate security solutions using the aspect framework. This entailed research into the classes of security issues that are present in software systems, as well as devising new solutions for these issues that would be most appropriate for the framework. One of the main facets of this thread was exploring the requirements for different types of vulnerabilities, ranging from simple buffer overruns to complex issues such as type safety. We built several aspects as part of this thread. The aspects ranged from ones that addressed the most common causes of security exploits, such as buffer overruns, race conditions and format strings, to higher-level and more complex issues such as type safety and event ordering.

In summary, the work performed under this contract has significantly advanced the state-of-the-art in the key areas of automating the addressing of security issues and the application of security solutions. We have developed and delivered a working and useable security framework along with several effective solutions (based on the framework) for addressing security issues. The key advantage of our prototype is that it

provides a very capable infrastructure that can be used to quickly address new security vulnerabilities using a process that integrates very easily with the developer's build process.

2. Summary of Motivation and Objectives

Over the past few years, software systems are increasingly being positioned as mission-critical elements of government and industry. Concurrently, and primarily because of this increase, there has been a heightened level of security-awareness amongst software vendors and consumers. This is especially true in government agencies and the military, where the usual security threats are compounded by the possibility of attacks against critical infrastructure and communications by hostile entities. This has given birth to a dire need for secure software systems and processes and techniques for achieving and assessing security concerns in software.

There are quite a few research efforts and tools available to detect vulnerabilities in applications [13,23,26]. However, current tools and techniques to address security issues in a structured, comprehensive manner are sadly lacking. The most common method of addressing security flaws in software systems is the "penetrate-and-patch" approach. This involves waiting till a security vulnerability is discovered in an application and then applying a narrowly focused patch that fixes the vulnerability in that location. Much has been written about the inadequacy of this technique [25]. However, there are two main shortcomings that are relevant to this paper. First, the very nature of this technique ensures that it is a reactive process – patches are applied to software usually after a flaw or an exploit has been discovered. Second, the technique usually focuses on fixing the discovered, endemic flaw, and not on addressing the issue at a global level.

In order to ensure more secure software systems, methodologies for addressing security concerns need to be more proactive and global. Security has to be considered a fundamental part of the system requirements, and as a result, the technique for addressing and applying security solutions to software systems should be an integral and seamless part of the development process. There have been some good efforts that work towards this goal. However, in order to have a usable and comprehensive technique, in addition to being proactive and global, it also needs to be flexible and adaptable enough to address new problems easily. This paper describes our effort to advance the research in this area and produce a tool to benefit the community.

The objectives of this effort were to devise a technique for addressing security issues in software systems that could be used as a framework for dealing with a wide range of concerns that affect the security of a system. The primary characteristics needed in the framework were:

- Proactive stance
- Global application
- Consistent implementation
- Adaptability
- Seamless integration

3. Detailed Final Report

3.1. Introduction

This report documents all technical work accomplished under the "*An Aspect-Oriented Security Assurance Solution*" project. Section 3.3 of this report, Technical Approach, contains a discussion on the problem addressed and the rationale for our approach. Section 3.4 of this report, System Prototype, details the research prototype system that we have developed in order to validate our approach. Section 3.5 is a cross-reference between the items covered in this report and the requirements outlined in the Statement of Work specified in the contract. Section 3.6 explains our validation and experimentation phase where we applied our approach to real world applications. The document also contains an appendix that contains the BNF grammar for the aspect language.

3.2. Innovations

In this project, we feel we have furthered the state-of-the-art in addressing security issues. Several new ideas and artifacts have been developed on this project

- New aspect language designed specifically for security
- Unified infrastructure/tool for addressing security issues in a global, consistent, flexible and modular fashion
    - Ability to address wide ranging set of issues with a single tool
- Set of aspects that address various security issues.
    - Achieved very quick turnaround for addressing new class of vulnerability (e.g. format string vulnerability)

3.3. Technical Approach

3.3.1. Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a relatively new programming paradigm that is based on the idea that computer systems are better programmed by separately specifying the various concerns (properties of interest) of a system, and then relying on underlying infrastructure to weave or compose them together into a coherent program. The foundation of AOP is the principle of "separation of concerns", where issues that crosscut (affect) the entire application are addressed in a modular manner by encapsulating them within an aspect. A simple example of that can be seen in Figure 1 below. This image shows a bar graph where each bar represents a module with the length of the bar indicating the size of the module. If in this particular application, if we wanted to log all file accesses, then in a traditional programming paradigm, we would have to find every place where we do file accesses and then insert appropriate code at each of those places to do logging. This can be a difficult and tedious task as shown by the image. The red colored portions of the bars represent all places where file accesses are being made. As

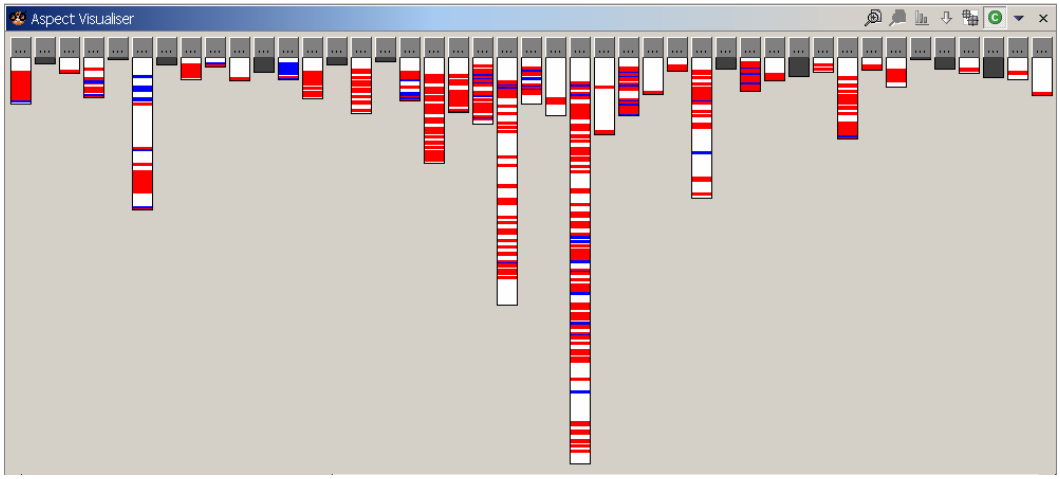


Figure 1: Tangled code required for adding logging functionality

can be seen, adding logging functionality would impact nearly every module in the application and cause much code tangling. AOP resolves this issue by allowing the developer to write a single, simple module that contains the logging code. This aspect would also contain information as to where the aspect is applicable (i.e. before all file accesses). This module is then composed with the rest of the code by the AOP infrastructure. Thus, AOP allows for clean, clear, global solutions to cross-cutting issues in software.

At the heart of AOP is the concept of an *aspect*. An aspect is a single entity that both embodies code to address a particular concern as well as how that aspect interacts with other code. In Figure 4, the module written was a simplified aspect to address the format string vulnerability. In addition to the aspects, we also need the aspect language in which the aspects are written and the aspect weaver, which composes the aspects into the rest of the code.

The following sections will talk more specifically about the related work in this area, the reason why security and AOP are a good combination and what we did on this project.

3.3.2. Related Work

The concept of Aspect-Oriented Programming (AOP) [10,11] developed out of the need to address issues in software development that have not been adequately addressed by previous programming models such as the object-oriented programming paradigm. As mentioned previously, the premise behind AOP is that software systems have inherent features and functionality that cannot be adequately articulated using the hitherto traditional means of abstracting and modularizing information and algorithms. The work done by Gregor Kiczales et al ([15]) has been at the forefront of the AOP thrust. They developed AspectJ ([16]) as an aspect-oriented extension to Java. AspectJ allows for the development of aspects as modular units of cross-cutting functionality. Aspects are defined similarly to classes and can contain fields, initializers, and methods, but contain

the code that might normally be scattered throughout the implementation in a traditional object-oriented design. Experimentation on the usefulness of AspectJ has been described in [19,26].

There has also been work done to provide AOP frameworks for other languages. AspectC [5] is an extension to C that is being used to provide separation of concerns in operating systems. Similarly, AspectC++[21] and AspectC# [17] are AOP extensions to the C++ and C# languages, respectively. In [3] a version of AOP is presented that is meant to be used with the Smalltalk language. This work describes an approach for dynamic aspect weaving in which the class that is to have aspects added to it is not modified at all. Instead, subclasses, which will implement the aspects, are created for the class. There is also an effort to use AOP to insert assertions statically in order to improve program robustness ([12]).

Like AOP, there are other similar techniques being worked on to address similar issues. The concept of Hyperspaces ([20,22]) is based on the notion that software systems have multiple dimensions and that each dimension must be considered in order to prevent concerns that cannot be represented in the major dimension from being scattered throughout the program. The Hyper/J tool puts into practice the hyperspaces paradigm by allowing the definition and composition of hyperslices to form a complete system. Similarly, the Adaptive Programming work by Lieberherr et al ([18]), addresses the fact that a certain piece of functionality that is to be included in a system ends up being spread over multiple classes. Using constructs called *adaptive methods*, however, the functionality that would normally be scattered throughout the code can be neatly encapsulated.

In addition to the various aspect-oriented programming related efforts, some of the efforts in the field of software security are also relevant. As mentioned previously, the Guardian series of security solutions by Wirex [7,8] is an interesting approach where additions are made to the compiler to enhance the security of the source code along specific paths. Other approaches to providing security to a software system include enhancing the language itself, as done in [14] and [9], both of which are safe versions of the C programming language.

The efforts described above have objectives that differ significantly from those of this project. All the AOP-related works are focused on creating generic extensions to various languages in order to implement a system that would allow AOP paradigm to be used. This holds for the Hyperspaces and Adaptive programming efforts, too. The work by Wirex, while being focused on security, does not offer an adaptive, comprehensive framework within which security solutions can be implemented. Finally, the language-based solutions offered by Vault and Cyclone are limited in the range of security issues they can address due to the fact that they have to work within the confines of the language.

The work we have done is unique in that we know of no other research effort that is treating security as a cross-cutting concern and focusing on building an AOP system for addressing security issues.

3.3.3. Security as a Cross-Cutting Concern

Security has always been a system wide concern, where the system is only as secure as its weakest link. In the case of a software system, this means that every module and code block needs to have the same attention paid to security and a consistent application of security solutions.

Consider one of the simpler security flaws that appear frequently in software systems: the buffer overrun. The common case here is that a stack buffer gets overwritten and overrun such that an arbitrary string can be executed. The usual procedure for fixing such vulnerability is to patch the immediate problem by ensuring that the data written to the buffer is less than the buffer size. This is fine and works in that the current problem is mitigated. However, this "*penetrate-and-patch*" approach does not work well in the long-term. The same user input that was used to overrun the buffer in this case is probably also being copied and transferred to other buffers along one of the many data flow paths that involve it. This may happen in the same module or another module of the application, or possibly in a library that the application uses. Such scenarios are commonplace and occur with increasing frequency in today's climate of modularized software systems. This problem can be alleviated by the use of security techniques that can be applied to the software system in a global fashion.

A related issue is that of implementing the security solutions in a consistent fashion. In today's climate of large, geographically diverse software development teams, it is difficult to ensure uniformity in the implementation of various features, including

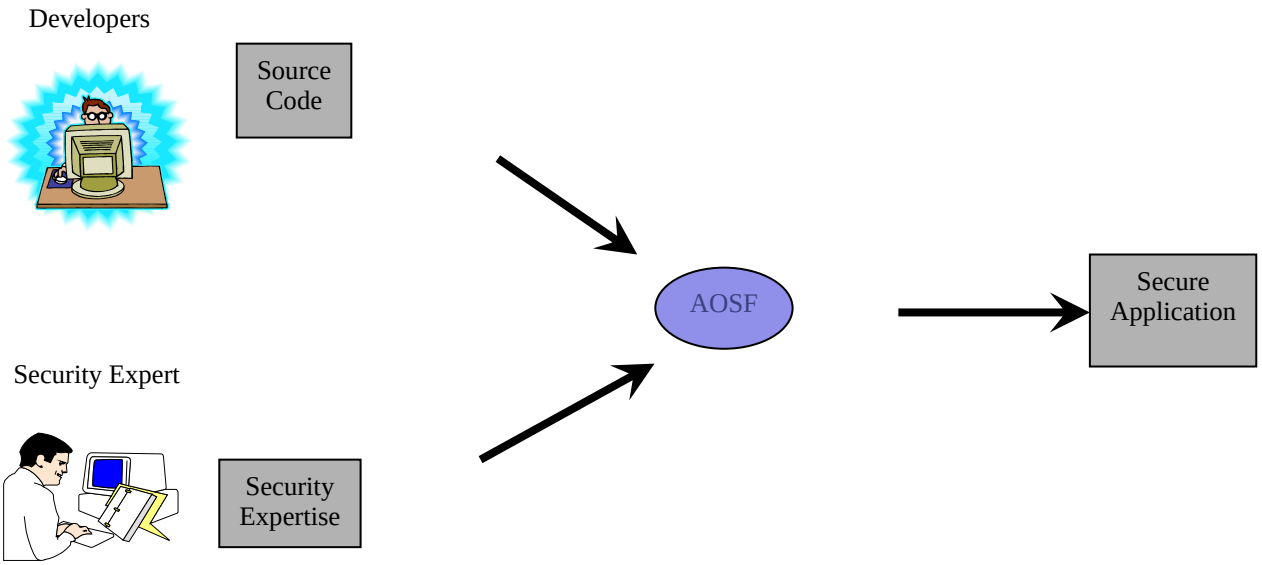


Figure 2: Separation of concerns for security

security concerns. An automated technique for addressing security concerns could considerably improve the degree to which the implementations conform to a standard level.

Considering the pervasiveness of security concerns within the code, established programming models such as procedural and object-oriented programming prove to be

insufficient to provide the right level of abstraction. In order to provide an appropriate foundation upon which security concerns can be considered in an appropriate fashion, it is necessary to recognize and view security as a cross-cutting concern.

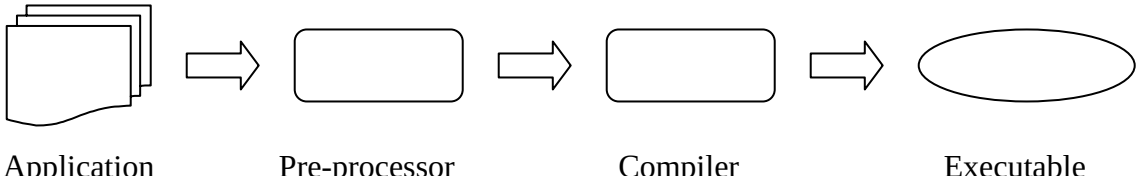
The framework developed as part of our effort uses the aspect-oriented programming paradigm with a view to treating security as a basic concern that permeates the whole software system.

3.4. System Prototype: The Aspect-Oriented Security Framework

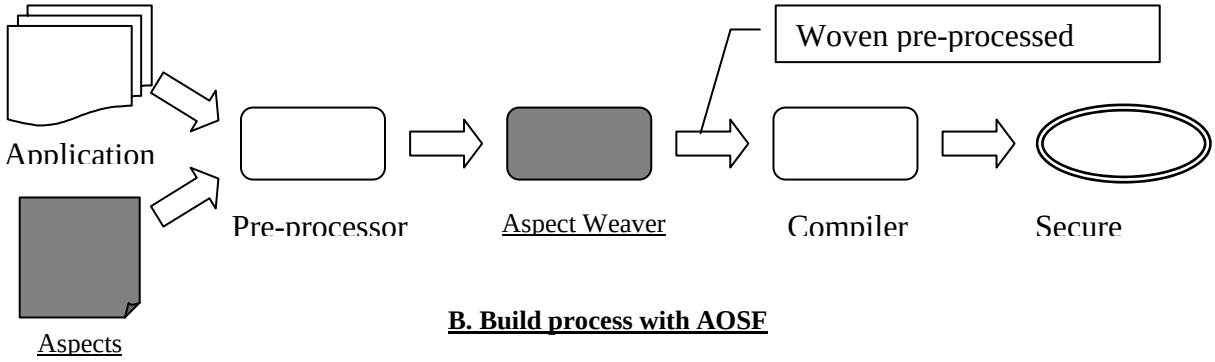
The security framework we have implemented, the Aspect-Oriented Security Framework (AOSF), is simple, flexible and intended to integrate seamlessly into the build process. The current implementation works with the C language, but we feel that the framework can be extended to other languages.

The central feature of this framework is the ability to encode security solutions as separate entities called *aspects*. The aspects are essentially recipes in a templatized form that provide generic instructions for performing code transformations in order to achieve a particular objective. In addition to providing the recipe, the aspects also provide a location guide that specifies where and under which code contexts the aspect should be used. Thus, the aspects define a complete security solution by detailing the *what*, the *why* and the *how*. In order to delineate these concepts in a generic fashion the aspects are written in a new language, called the aspect language. The aspect language is a superset of the application language in order to provide an easy transition between the application language and the aspect language for the aspect writers. Aspects are intended to be written primarily by security analysts, but can be used by application developers as well. The AOSF uses aspects as a central feature around which to build an integrated foundation for addressing security issues.

As can be seen from Figure 3A, a normal build process consists of the various source files being pre-processed and then compiled and linked to form an executable. The AOSF process flow is very similar to the normal flow, except that it adds an extra step to the process. As shown in Figure 3B, the process starts off with the application source code being pre-processed. In this case, we also pre-process the *aspects* that we want to integrate with the application. Once the application source code and the relevant aspects have been pre-processed, they are sent to the *aspect weaver*. The aspect weaver is the core component of the framework. It takes the application source code and builds an internal in-memory representation of it. The internal representation of the source code is a data structure that is structured to include all the necessary syntactic and semantic knowledge for enabling code transformations. Once this data structure is built, the weaver takes each aspect and, based on the directions in there, maps the aspect to locations in the code where it can be applied. It then takes the aspect code and creates a code segment that encapsulates the aspect directives and is specific to that code location. Thus, the aspect weaver creates solutions that are context-dependent and location-specific from the generic aspect directives using code transformations



A. Normal build process



B. Build process with AOSF

Figure 3: AOSF Process Flow

This process is repeated for each aspect that needs to be applied to the application source code. Once all aspects and their target locations have been transformed, the aspect weaver begins to convert the in-memory representation back into compilable code. The output of the weaver is akin to the output from a pre-processor and is thus designed to be fed straight into the compiler.

The process detailed above is simple and requires a minimal effort on the part of the developers in order to use it.

3.4.1. Framework Characteristics

This AOSF provides a broad base upon which to build solutions to security issues. In addition, it also fulfills the needs of a security framework, as listed in Section 2.

Proactive Stance

The current "penetrate-and-patch" approach to security has many problems. Chief among these is that it is a reactive approach, and as such is always behind the attack curve. The framework described in this paper is designed to be used as part of the development process such that security can be applied to the software system by default. This facilitates the use of the framework as an integrated stage of the software development process where security is not just an afterthought, but is applied with a proactive view.

Global Application

Security is a "whole program" issue: each and every part of a software system needs to be strengthened from a security perspective. Any part that is not adequately secured will

```
1   aspect formatStringAspect {
2
3   int count_parameters() {
4           …
5   }
6   int count_fields( const char * format ) {
7           …
8   }
9   funcCall< int sprintf(char * str, const char * format, ...) >  {
10          before {
11                  if( count_parameters() != (count_fields(format)+2) ) {
12                          fprintf( stderr, "FormatStringAspect error: bad string\n" );
13                          return -1;
14                  }
15          }
16  } /* funcCall */
17  } /* formatStringAspect */
```

Figure 4: Example Aspect

result in the security of the entire system being decreased. Thus, it is critical that any security measures that are applied to a system be applied across the board. By treating security as a cross-cutting concern, the AOSF allows security analysts to apply security solutions globally, while giving them the flexibility to focus on particular parts of the system if needed.

Consistent Application

The global application of security measures is an important step in securing systems. However, in order to achieve the desired effect, it is essential that the security solutions be implemented in a consistent fashion. This is a growing problem in the current climate of large, geographically diverse development teams. The AOSF alleviates this concern by automating the process of integrating the security solutions into the software system. This eliminates the problem of inconsistent implementations of the same solution.

Adaptability

The framework provides a full-featured transformation engine and an expressive but simple language for encoding generic directives for security solutions. These ensure that the framework can be used to implement a wide-ranging set of security solutions. In addition, it also eases the process of responding to and addressing new security issues that may be discovered.

Seamless Integration

One of the strengths of the AOSF tool is the ease with which it can be integrated into the build process. Since the core of the framework, the weaver, can masquerade as a

second-stage pre-processor, the only change that the developer has to make is change the compiler in the application Makefile. For example, in a typical UNIX environment, the modification is limited to changing:

CC=gcc

to

CC="aspectweaver <weaver options> gcc"

The aspect weaver, in this case, will first call the pre-processor to process the source code and the aspects. It takes the output from this stage and weaves the aspects into the source code as described above. Finally, it calls the specified compiler on the output of the weaving stage in order to get compiled code. This ensures that the AOSF will have an easy integration path into the build process, and is more likely to be used by developers.

3.4.2. High-Level Architecture

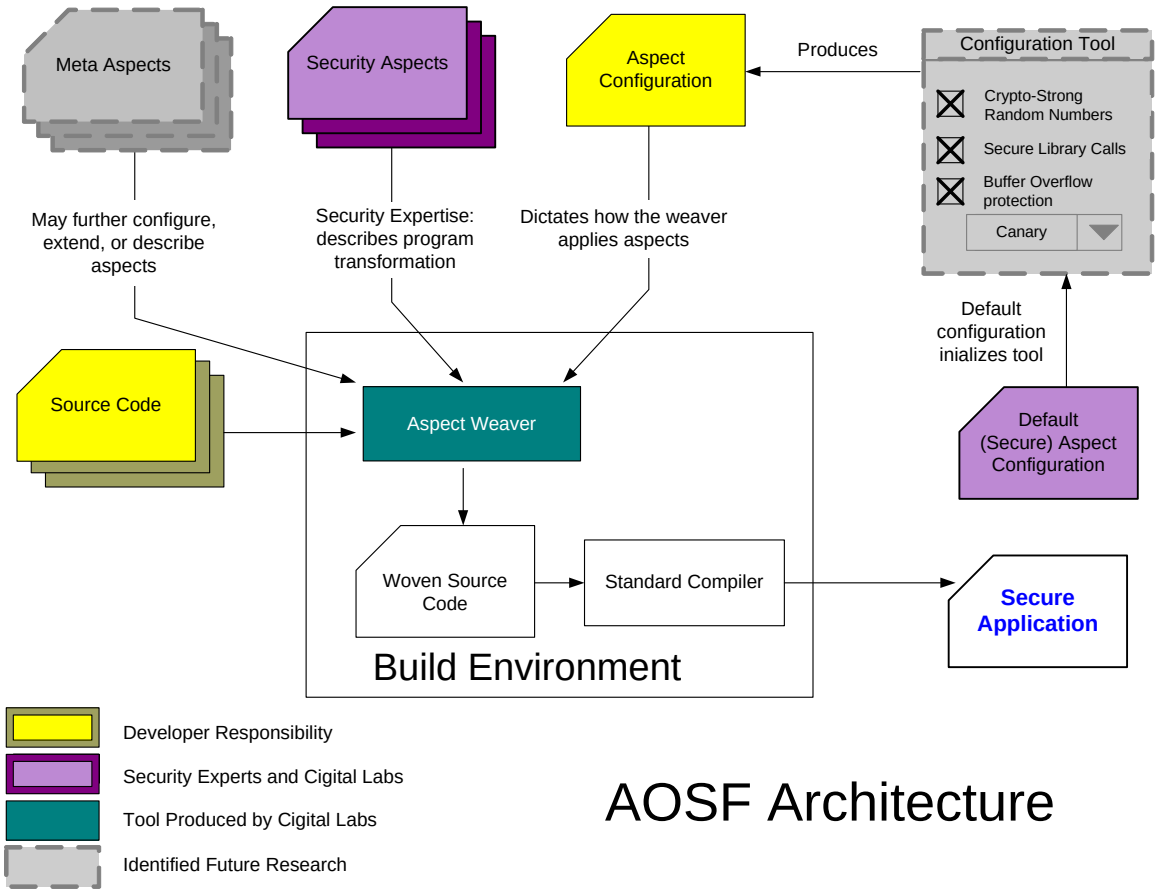


Figure 5: AOSF High Level Architecture

3.4.3. Aspect Language (CSAW)

The aspect language that we designed was created specifically with security issues in mind. This meant assessing the needs of various security issues and creating a language that would meet its purpose but be simple enough for it to be useable. The language that we arrived at is a minimalist language that is a small superset of the C language. For an example of the aspect language, see Figure 4.

3.4.3.1. Structure

At its simplest an aspect begins with the keyword aspect and is followed by that aspect's identifier, its name. The aspect's implementation is defined within the subsequent '{' and '}' braces. The body of a simple aspect could be comprised solely of location declarations. Think of a location declaration as a tuple.

An Aspect's Constituents

An aspect potentially contains two different types of things: C-style definitions and location declarations.

Any C-style definitions present in an aspect must come before the first location declaration and represent functions and variables valid within the scope of the aspect. Functions defined here are callable from within the aspect and variables declared within are within scope throughout the aspect's life-they can be referenced from location declarations. Variables serve to hold aspect state and functions will serve to support the aspect.

Location Declarations

Following the aspect's definitions are its location declarations. Each describes one or more points within a program identified requiring modification. Each location definition begins with a point quantifier, which is optionally narrowed by a area qualifier. Code quantifiers follow each point qualifier. Each construct's purpose is described below.

Point Qualifiers

A point qualifier describes where modifications will be done. Currently, function calls and definitions can serve as point qualifiers. One may also use a label or a range between two marking labels as point qualifiers. A point qualifier looks like a templated function. Below is an example of a function call's point qualifier:

funcCall<int main(int argc, char** argv)>

specifies a cut on the common main() we're all familiar with. More information on point qualifier is available in Section 3.4.3.2.

Area Qualifier

An area qualifier changes the scope of the C program under which the point qualifier it is specified after should be matched. An excellent description of area qualifiers as well as further discussion of specific qualifiers exists in Section 3.4.3.2. Area qualifiers currently

take one of three forms, allowing for narrowing location matching to inside a function, module, or aspect. Multiple area qualifiers can be used on a single point qualifier. In this case each is concatenated using the boolean operations and, or, and not. An example of an area qualifier follows:

funcCall<int main(int argc, char**Arv)> : inModule<MyCFile>

The above specifies that the main() function should only be matched if in MyCFile.

Code Quantifier
Having described where modifications are to be made, code quantifiers describe what should be done. This occurs within the '{}' body of the location declaration. Specific code quantifiers come in the form of three types of blocks: before, after, and replace. The specific semantics of each is explained in Section 3.4.3.2. Context gathering variables can be referenced within code quantifiers. These give aspect code useful information about their current context and are explained in Section 3.4.3.2. They currently include information about the current aspect, C module, C function name, C line number, and the return value of the current C function.

3.4.3.2. Semantics

This section endeavors to explain the semantics behind the important syntactic constructs in the C-Saw language.

Point Qualifiers
Point Qualifiers specify the point or series of points that a location will refer to. These can in general contain wildcards and variable names such that the parameters at certain locations can be used in inserted code.

Function Definition funcDef < Signature >
This will set the location to the body of the function definition specified by function signature *Signature*. Names from the Location-Dec, types, or wildcards can be used to bind parameters. The return value can only be bound by type or wildcard. If the value for the return parameter is needed in the after code section, it can be bound with the appropriate after method. Additionally, if parameters passed into the function are changed by the function call, the parameters that function was called with are available in the before section and the changed parameters are available in the after section (as long as the values were bound to names in the Location-Dec declaration). Additionally, the code specified in the after section will be executed as if it were inserted before the return statement(s) of the function.

Function Call funcCall < Signature >
This will set the location to calls of the function specified by the signature *Signature*. Names from the Location-Dec, types, or wildcards can be used to bind parameters. The return value can only be bound by type or wildcard. If the value for the return parameter is needed in the after code section, it can be bound with the appropriate after method. The

value bound to the parameters here differ from the PQ-FuncDef element in that if the function changes the parameters in the function definition, the original values used to call the function will be available in the after code section.

Label Statement labelMatch < Wild-Identifier >
This will set the location to the label statement marked by Label in the source code. This is exclusive of the statements before and inclusive of the statement associated with the label. If the statement associated with the label (or any statement contained therein) is control-flow altering (return or goto), the code inserted after this point qualifier will be inserted before the control-flow altering statement as well as after the labeled statement. Wildcards are allowed on the name of the label.

Label Range labelRange < Identifier , Identifier >
This will set the location to the code between the two source code jump labels given (inclusive of the jump labels). This essentially means that code inserted in a before or after code quantifier section will appear before the first jump label and after the second jump label, respectively. Wildcards are not allowed, but there is an underscored integer notation used to specify two label ranges in the same level of nesting. This essentially means that not only will the specific labels stated in this point qualifier be matched, but also the same label pair with an underscore and integer appended to each label. The only constraint to this is that the integer must be the same for both items in the pair. Thus if my point qualifier specifies (foo,bar) as the labels on which to cut, then source blocks labeled (foo,bar), (foo_1,bar_1), and (foo_99,bar_99) will each match, but (foo_1,bar_99) will not.

Area Qualifiers
Area Qualifiers may not be used by themselves to delineate a location. They are used only to scope the definition for a Point Qualifier. By default, Point Qualifiers operate only on code in modules and not on other Aspects. This can be explicitly changed by the use of the appropriate Area Qualifier element.

Module inModule < Wild-Identifier >
This will limit the location to only the matches for a Point Qualifier that are within the specified module Module-Name. Wildcards can be used. In C, the module name is given either by the filename containing the source as offset from the current dir, or by the fully qualified path and filename. For example, if the code was in /home/bob/proj/myModule.c and the current dir was /home/bob, Module-Name would be either ``proj/myModule.c" or ``/home/bob/proj/myModule.c".

Function inFunc < Signature >
This will limit the location to only the function definition specified by the signature Strict-Signature. When using this qualifier, note that combination of this element with the PQ-FuncDecl or the PQ-FuncDef elements will lead to an empty location set since there will not be any definitions or declarations in the body of the function Strict-Signature.

Aspect inAspect < Wild-Identifier >
This will expand the location to matches for a Point Qualifier that are within the specified aspect name Aspect-Name. Wildcards can be used. The aspect name is given either by the simple name if in the name is locally resolvable (same package or imported) or by the fully qualified aspect name with preceding package name(s).

Context Gatherers
These are special names that can be used in code sections to access useful context information. Each is given with the C type to which it conforms.

String Literal _modulename
Usage of this name in the Code Quantifier sections should be exactly as one would use a string literal in C code. The name itself is replaced with a string literal that contains the simple name of the module in which this location is matched. This value does not have any path information included.

String Literal _funcname
Usage of this name in the Code Quantifier sections should be exactly as one would use a string literal in C code. The name itself is replaced with a string literal that contains the name of the function in which this location is matched. The value is the simple identifier name of the function and does not have any information about the functions parameters or return type. This value is the empty string "" if the location is not in a function.

String Literal _aspectname
Usage of this name in the Code Quantifier sections should be exactly as one would use a string literal in C code. The name itself is replaced with a string literal that contains the aspect name in which this location is defined.

Integer Constant _lineno
Usage of this name in the Code Quantifier sections should be exactly as one would use an integer constant in C code. The name itself is replaced with an integer constant that contains the line number on which this location is matched.

Typed Variable _retval
Usage of this name in the Code Quantifier sections should be exactly as one would use a variable of same type as the return value of the function matched in C code. The name itself is uninitialized if referenced in any before or replace Code Quantifier section. Assignments to this name are meaningless in these sections as well. In any after sections, however, this name is set to the return value of the function. Assignments to it here are subsequently treated as the value to return. This name will be set to an integer equal to zero if the location being cut on does not have a return value.

Code Quantifiers
Code Quantifier constructs are used in conjunction with a fully defined location in order to specify the code that needs added/augmented specific to the location. Each instantiation of a location may have only one of each of the following code quantifiers.

Before <before Cmpd-Stmt >
This construct allows for code to be inserted before a match of the predefined location. The Cmpd-Stmt section contains the code that will be inserted at the location. All names referenced in the Cmpd-Stmt section must either be variables bound by the location declaration, a symbol declared in the aspect itself, a symbol declared in the location definition, a symbol declared in the Cmpd-Stmt, a valid context gathering identifier, or an escaped identifier to be verified in the context in which the code will be inserted. Multiple before sections for a location are inserted sequentially, i.e. the order specified.

After < after Cmpd-Stmt >
This construct allows for code to be inserted after a match of the predefined location. The Cmpd-Stmt section contains the code that will be inserted at the location. All names referenced in the Cmpd-Stmt section must either be variables bound by the location declaration, a symbol declared in the aspect itself, a symbol declared in the location definition, a symbol declared in the Cmpd-Stmt, a valid context gathering identifier, or an escaped identifier to be verified in the context in which the code will be inserted. Multiple after sections for a location are inserted sequentially, i.e. the order specified.

Replace < replace Cmpd-Stmt >
This construct allows for code to be inserted instead of the code in the match of the predefined location. The Cmpd-Stmt section contains the code that will be inserted at the location. All names referenced in the Cmpd-Stmt section must either be variables bound by the location declaration, a symbol declared in the aspect itself, a symbol declared in the location definition, a symbol declared in the Cmpd-Stmt, a valid context gathering identifier, or an escaped identifier to be verified in the context in which the code will be inserted. Multiple replace sections for a location replace whatever was previously inserted, thus the last replace section specified will be the only one present when weaving is done.

3.4.4. Aspect Weaver

This section hopes to clarify the inner workings of the weaver in a somewhat sequential fashion. While many of the functions of the weaver do indeed cross the boundaries between stages, this document will hopefully clarify the general sequence and perhaps not the actual implementation. Two passes over the document should probably be taken in order to get a better understanding.

3.4.4.1. Warp

A Warp is a weaving term referring to the vertical strands on a loom that are threaded by the Woof. In our case, this refers to the C file that has been parsed to the internal representation.

3.4.4.2. Woof

A Woof is a weaving term to refer to the strands attached to the shuttle that is passed through the vertical Warp strands. The Woof represents the aspect that will be passed over the C internal representation (IR) to produce weaver output.

3.4.4.3. Lexer

There is one lex lexer used to lex both C and C-Saw code. The lexer employs multiple states so as to be compliant with both languages. The C_MODE in the lexer is used for plain C code and the CSAW_C_MODE is used for C-Saw code. This is due to the addition of language keywords that should only be tokenized as such in the C-Saw context. There is an additional mode called C_MODE_WILD which is used to parse C code that may contain wild identifiers (identifiers containing question marks) and the any keyword. The switching between these three states is controlled by the parser. The lexer also maintains a hashtable that contains type-names that have been assigned by use of typedef constructs. This table is kept because an identifier and an identifier that has been made a typename must be tokenized as two different things. Type-names are added to this table by the parser once a particular construct has been recognized to contain a typedef declaration. The typedef constructs cause C to be non-context-free, but this parser-lexer feedback alleviates this problem.

3.4.4.4. Parser

There is one yacc parser used for both C and C-Saw. Its state is controlled by two functions that can be called to parse either a C or C-Saw file and return either a Warp, or a Woof. The initial state switch for the parser is accomplished by setting a variable that the lexer reads and then returns as an initial dummy token. In general, the parser has three main purposes. The first is building the internal representation. The second is adding TypeInfo objects to nodes that may have ``interesting" type information. The last is alerting the Woof whenever a node that may be cut on is created.

3.4.4.5. Weaver Sequence

Stage 1: Creation of Warp and Woof

The weaver first endeavors to read in and parse the C-Saw file(s) passed in from the command line. As each Woof instance is created, the Woof class retains a static reference to the instance. This is done to facilitate the aspects being alerted of nodes in the C internal representation that they may be interested in cutting on. After this is completed, the parse method is called on the C file.
As the C File is parsed, the standard typedef feedback to the lexer is made along with the standard invocations of the TypeInfoVisitor to build TypeInfo objects for relevant nodes.

Additionally, as the file gets parsed, calls are made to the static function Woof::alert() when a node is created on which there could possibly be a cut. The Woof class contains a static list of references to instances, so it calls everyone on the chain with the parameter passed in to the Woof::alert() method (the parameter is a pointer to the node just created by the parser). Each Woof instance now has the chance to determine if it is interested in the node with which its callback() method was invoked. The callback() method for each instance determines if it is interested in the node by invoking a match() function on each of the location declarations in the instance. If a Woof instance is interested it keeps a tuple of the location declaration and the internal representation node, and if not interested, it does nothing.
The output of this phase in the process is a Warp instance and one or more Woof instances that each knows what part(s) of the Woof instance in which each has interest.

Stage 2: Weaving

The constraint on this phase is simply to output the same Warp instance and Woof instances that went into this phase. Keep in mind that after this phase, the parse time registrations of nodes to the static Woof method are no longer valid, thus in order to weave again, a pass must be made over each Woof to clear its list of tuples (described briefly above) and then a pass must be made over the Warp to register any interesting nodes.

Stage 3: Code Generation

Code generation is the simplest part of the whole system described here. This phase only uses the Warp. A print() instance method simply creates StringVisitor and drops it on the root node of the tree contained in the Warp. The StringVisitor visits the tree depth first and concatenates characters to an internal string until the whole tree has been visited. When this is completed, the string is sent to stdout. Formatting in the output text is minimal so send output to indent or some similar code formatter is recommended.

3.4.5. Security Aspects

The Aspect Oriented Security Framework rests on the capability to create, parse, and weave aspects. The aspects themselves are expressed in our aspect language, called C-Saw, which is a small superset of C Language (see Figure 4 for an example aspect).

A large amount of effort on this project has been dedicated to the research, development, debugging and validation of a library of Aspects designed to improve the security of C programs.

3.4.5.1. Low Level Aspects

Much of our early work in the project was dedicated to creating relatively simple aspects to address legion security vulnerabilities. For instance, we adapted the well-

known canary protection technique into a simple aspect. These aspects typically defend against a specific type of attack but lack some generality.

Environment Aspect

A number of programs can be exploited through the use of clever or surprising construction of their environment variables. The Environment Aspect protects a program against exploitation though its environment variables. It is implemented with a point-cut at the declaration of main(), the standard entry point to C programs. In the before block of the point-cut, before any other code is executed, all of the original environment variables are cleared. Then, a few critical environment variables are written back with safe values.

This aspect protects against a few potential attacks against setuid programs. First of all, a setuid program probably shouldn't ever use environment variables for configuration. An attacker has total control over them and they are a potential source of buffer overflows. An even more serious problem stems from the fact that the environment gets passed to any programs spawned by the setuid program. Even though the setuid may not be exploitable by environment variables, a subshell spawned by popen(3) might be (especially by IFS - internal file separators, which is used by shells to tokenize the command line).

Of course, the Environment Aspect is probably not appropriate for programs that won't have the setuid bit set.

Format String Aspect

The Format String Aspect protects against the relatively new format string attacks. It uses point-cuts at any call-sites to the printf(3) family of functions to ensure that the number of parameters passed to a printf matches the number of fields specified in the format string. Essentially, the before block of a wrapped printf(3) function queries the actual number of parameters passed and then counts the number of fields described by the format string. If these are not equal, the operation is not allowed to proceed.

Canary Protection Aspect

One of the most straightforward uses of the aspect language is to craft a canary transformation to protect each stack frame from the dreaded buffer overflow exploit. A canary value, when placed between a stack frame's return address pointer and an overflowed buffer, will be corrupted in the event of a buffer overflow exploit. By randomizing this canary value, it becomes possible to abort a program before returning from a function in the event the canary is corrupted. Unfortunately, this approach will not protect against heap overflows or attempts to overflow stack variables.

Parameter Sanitization Aspect

Parameter sanitization is a broad topic that we have only scratched on this project. A simple input sanitization aspect was written which checks the parameters to the set of UNIX system calls which spawn child processes. The parameters to these system calls were checked for certain shell meta-characters that often cause children to go awry in CGI programs. As an example, oftentimes form data from a web page is fed directly to a CGI bin program that calls a subprogram to perform some actions on the form data.

When the form data contains a pipe symbol or a semi-colon, the subprogram can be hijacked. Our parameter sanitization prevents this from occurring.

Encrypted Communication Aspect

Programmers attempting to make use of encryption often make any of a number of mistakes. We put together a simple aspect that ensures that all communication is encrypted, that the underlying encryption libraries are used correctly, and that the keys in use are stored securely.

Time of Check to Time of Use Aspect

The Time of Check to Time of Use Aspect protects against many of the Time To Check To Time Of Use (TOCTTOU) attacks which plague attempts by UNIX programs to safely make use of the /tmp directory. We make certain that multiple attempts to reference a file using its filename is translated into one attempt to reference a file using its filename while the remainder reference the file using a file descriptor.

TOCTTOU vulnerabilities exist when programs attempt to use a filename as a reference to a file on disk, usually in the /tmp directory. Essentially, between one file reference and the next, some attacker has interposed himself. The classic race occurs between a stat(2), which checks for permission, and an open(2). In between these two calls, an attacker replaces the file in question with a symbolic link to somewhere important. Because the aspect weaver has rewritten these calls in terms of file descriptors, the attacker's newly created file is not referenced at all.

Several limitations are written into the current implementation of this aspect. First of all, it only checks for stat(2)/open(2) race conditions. Other classic races, such as access(2)/open(2) race conditions are permitted. Furthermore, even with respect to the stat(2)/open(2) races we make the assumption that one stat(2) is followed by one open(2) and then one close(2). Calls to dup(2) or multiple calls to open(2) will bypass our checks.

3.4.5.2. High Level Aspects

Our high level aspects grew out of a desire to solve more general and less implementation specific software security faults. For instance, we originally wrote a low level aspect that could correct Time to Check to Time of Use failures in source code. We eventually generalized this idea in the form of an Event Ordering Aspect, which allowed programmers to specify in a high level fashion the correct usage patterns of arbitrary functions.

Signal Race Condition Aspect

Wu-ftpd was once vulnerable to a signal handling race condition, and some related info could be found at http://www.cert.org/advisories/CA-1997-16.html and http://www.landfield.com/wu-ftpd/mail-archive/wu-ftpd/1997/Jan/0004.html. While reviewing the source code of wu-ftpd 2.6.1, we felt it interesting to see the precautions the authors took to delay signal delivery for privileged blocks, because 'labelMatch' and 'labelRange' in C-saw were designed exactly for this type of situation (see http://sandbox.rstcorp.com:8080/pipermail/aop/2001-August/000322.html.) In doing a 'grep' for 'NEED_SIGFIX', 'delay_signaling', and 'enable_signaling' with the wu-ftpd source code, this becomes clear.

With the current capability of the weaver, we found it simple to take a 'remove-then-weave' approach to demonstrate the power of AOP in preventing signal handling race conditions. General arguments of the benefits of AOP apply here.

Wu-ftp is not alone. Recently, Procmail was also reported to have several potential race condition vulnerabilities (http://www.securityfocus.com/cgi-bin/vulns-item.pl?section=discussion&id=30 71). Sendmail was reported to be susceptible to possible race condition vulnerabilities (<http://www.securityfocus.com/cgi-bin/vulns-item.pl?section=discussion&id=27> 94). Michal Zaleski at Bindview wrote a paper titled "Delivering Signals for Fun and Profit", which can be found at http://razor.bindview.com/publish/papers/signals.html, he concluded the paper by saying that,

> "This is a very complex and difficult task. There are at least three aspects of this:
>
> - Using reentrant-safe libcalls in signal handlers only. This would require major rewrites of numerous programs. Another half-solution is to implement a wrapper around every insecure libcall used, having special global flag checked to avoid re-entry,
> - Blocking signal delivery during all non-atomic operations and/or constructing signal handlers in the way that would not rely on internal program state (e.g. unconditional setting of specific flag and nothing else),
> - Blocking signal delivery in signal handlers."

It is of course pleasing to see him use the word aspect. Using AOP, the task is not that "complex and difficult". The second aspect is actually similar to what was done in wu-ftpd, as described earlier, it can be solved, and an aspect designed for the 'remove-then-weave' approach could also be applied to other programs. We have crafted a decent solution based on the third of the described approaches. Through the use of programmer-inserted labels, we prevent the delivery of signals while in signal handlers. Thus, use of this aspect can effectively prevent these kinds of exploits.

Event Ordering Aspect

The goal of the Event Ordering Aspect is to allow a user to fully specify the correct order in which a group of related functions may be called. This work is an extension of our original Race Condition Aspect which prevented Time of Check To Time of Use exploits. We felt these race conditions were part of a broader category of flaw related to improperly ordered system calls.

In describing this aspect, we are using letters to symbolize functions, you'll notice that the functions below are thinly disguised malloc (aka M), realloc(aka RF), free(aka F), calloc (aka C), and realloc(aka R) Although not shown in the example, the user must specify the prototypes for all functions referenced. The '|' is meant to be a logical OR. The error key word just indicates what to do if the expected order is not followed. We also allow the user to specify "Error: exit" or something to halt the program. Consider the following annotated example:

```
Order block: M, C, RF, R, F
        /* denote that we are about to specify a set of ordering rules for M, C, RF, R, F */
{

        Function: void F( ...p1, ... ) {
        /* Indicates F's p1 has to match with a ret. val of the following functions. This implies
        that one of RF, R, M, or C must have been called prior to F and the return val was
        stored in a list. Once a return val that matches p1 has been found it is removed from
        the list. The same is followed in the following function blocks */
                F-p1: RF-ret | R-ret | M-ret | C-ret
                Error: Log( "Error in operation order ...");
        }

        Function: void* RF( ... p1, ...) {
                RF-p1: RF-ret | R-ret | M-ret | C-ret| NULL
                Error: Log("error in operation order....")
        }

        Function: void* R( ... p1, ...) {
                R-p1: RF-ret | R-ret | M-ret | C-ret|NULL
                Error: Log("error in operation order....")
        }
}
```

Also, as specified above, all return values of M, C, R, and RF are stored in a list for future comparison; however we do not store null in the list if these functions return null. Additionally, we allow notation like: X-p1: M-ret!=NULL | C -ret!=NULL to indicate that p1 should match up with some non-null return value from a previous call to M or C (thus indicating that NULL should never be stored in the list)

Another ordering block follows that includes functions M, C, and X (where X is some random function).

```
Order block: M, C, X
{
        Function: void X (p1...) {
                X-p1: M-ret | C -ret
                Error:...
        }
        ...
}
```

Now with this order block, we again know we are storing return values of M and C, but in practice a new storage list is instantiated since this is a different ordering rule from the first example.

One thing we have not yet specified is the way to deal with cases where functions like malloc, realloc, etc. return null. This especially gets tricky for realloc, because if realloc can't accommodate the request it was given it leaves the original pointer sent in as a parameter unchanged. That means the pointer should still exist in the list after realloc returns in this case, however we are not sure how to indicate that it needs to be reinserted except by allowing the user to add in some random if statement. With reallocf, returning a null value is not a problem because it will automatically free the pointer in this case, and that pointer will already have been removed from the list to do the comparison. So let us examine error conditions in greater detail.

```
Order block:
{
  void Afunction() : A
  void Bfunction() : B

  ERROR: <user specifies some action - like log ordering violations>
  B: A
}
/* This example indicates that for every B executed, an A must have executed before it. If this
rule is violated, the Error statement will fire.

The generated aspect code would require function A to have an "after" block where A is
inserted into the list and B to have a "before" block where the list is checked to insure that an
A is in it, and if an A is there it is removed */

Order block:
{
void * malloc( int size ) : M
void * calloc( int size ) : C
void * reallocf( void * ptr, int size ) : Rf
void * realloc( void * ptr, int size ) : R
void free( void * ptr ) : F

ERROR: <blah, blah, blah>

F-ptr: (!NULL) && (M-ret || C-ret ||  Rf-ret || R-ret)
/* (ptr can't be null and must match one of the ret-vals of the functions on the right hand
side)*/

Rf-ptr: Rf-ret | R-ret | M-ret | C-ret| NULL

R-ptr: Rf-ret | R-ret | M-ret | C-ret| NULL

/* the ptr sent to R or Rf must match one of the return vals on the right hand side, but null is a
legal input parameter for R and Rf */
}
```

Of course, as stated above, we have a slight kink with realloc/reallocf. By default, whenever there is a match with something on the right hand side to the item on the left hand side (e.g a ret val of M matches a ptr of R), the matched item is removed from the list. This works out well when realloc or reallocf complete successfully (i.e. retval is not null) and when reallocf returns a NULL (because reallocf's implementation frees the param passed in when it can't successfully allocate the requested amount). However,

when realloc can't allocate the requested amount of space, it leaves the original pointer in the list. By default, the language always removes a list item that has matched with the left hand side of a rule. So we have added a way to denote the fact that in some cases, the item should NOT be removed even if it matches.

```
Order block:
{
        void * malloc( int size ) : M
        void * calloc( int size ) : C
        <ret-type> X(<param_type> p1) X

        ERROR: <whatever>

        X-p1: M-ret | C -ret
}
```

Here, based on the right hand side of the expression, we know that return values of M and C must be inserted in the list (and of course, this is a different list from the previous rule) and when X is called, its p1 has to match with an item in the list.

Future Event Ordering Aspect

We feel that our current Function Ordering Aspect has some room for improvement. First of all, the current language cannot express an ordering trace in cases a single function must be called repeatedly. Secondly, the notation could be much more terse. Thirdly, the way in which execution traces are expressed is coupled too tightly to the aspect language. In short, although this was supposed to be an example of a meta-aspect, it is not, as it stands, very meta. Therefore, we present the following possible way of improving it in the future.

```
    void * malloc( int size ) : M : NULL
    void * realloc( void * ptr, int size ) : R : NULL
    void free( void * ptr ) : F : void

    ( M<r> | R<r> ) ( F<p1> | R<p1> )
```

Consider once again the case of malloc, realloc, free.

Let us examine this notation. The prototype-like constructs at the top are unfortunate but necessary. Currently, they encode three pieces of information. The first, the prototype itself, is an essential ingredient for building the subsequent aspect file. The second piece of information is a shorthand representation of the function to be used below. The third piece of information is the list of error return values the function may return.

The regular expression in the example should be read something like this: Either a call to malloc (M in the shorthand) or a call to realloc (R in the shorthand) causes this regular expression to fire. The return values for either of these functions shall act as a key

to match against further calls (the appended <r> in the expression). Further, either a call to free (F in the shorthand) or a call to realloc (R in the shorthand) shall cause a transition and the completion of the regular expression, assuming they are called such that their first parameter matches the key (the appended <p1> in the expression).

Let us also examine possible error conditions in some detail. In the default case, when a function fails, the REGEX should not attempt to make a transition. We can construct additional notation to allow the user to account for cases where the converse is true.

Consider this slightly simplified variation of malloc, realloc, and free.

```
void * malloc( int size ) : M : NULL
void * realloc( void * ptr, int size ) : R : NULL
void free( void * ptr ) : F : void

M R F
```

In the rule we have just constructed, malloc must be followed by realloc must be followed by free. But what happens if the call to realloc fails (returns NULL)? To a certain degree, it just depends on the function. In this case, the REGEX should probably just remain in the old state.

So, legitimate sequences would include:

M(succeeds) R(succeeds) F(succeeds)

M(succeeds) R(fails) R(fails) R(succeeds) F(succeeds)

but not:

M(succeeds) R(fails) F(succeeds)

M(fails) R(succeeds) F(succeeds) //note the REGEX isn't successfully entered

It is possible, though, that when a function fails the REGEX should complete (a real-life example of this is reallocf, which frees memory on failure). It shouldn't be too difficult to do add some notation to the regular expression which will flag these kinds of functions.

Type Safety Aspect

Type safety enforcement can be an important tool in writing secure and correct programs. We have used the AOP weaver to collect information which can be used at run-time to query type information (RTTI). This information can be queried in various situations to enforce some aspects of type safety.

At program load time (strictly speaking, the very beginning of program run time), we construct offset maps of local variables for all functions and a similar map for global variables. These maps are used by the RTTI query engine to look up the type information which corresponds to a given memory address. The local variable maps are generated through the use of struct'ed stack frames.

The simplest use of RTTI is very simple and very resource inexpensive. The weaver is used to rewrite calls to the strcpy() family of functions such that they query the types of the parameters. Since this type information includes the size of buffers, it can be used to

ensure that the source buffer is shorter than the destination buffer. This much we have actually implemented.

A more complicated use of RTTI is correspondingly more expensive. Cuts must be made at assignments in order to check the compatibility of the rvalue type with the lvalue type. A number of techniques may be applied in order to determine the compatibility of types. A simple idea may be to allow a buffer be re-interpreted as only one different type. Alternately, a configuration file might be used by the developer to describe type compatibility.

We probably need to perform at least some static analysis in order to reduce redundant checking.

3.5. Statement of Work References

This section references each of the items in the statement of work, and identifies sections of this report that address the completion of each task.

Tasks/Technical Requirements.

4.1 The contractor shall accomplish the following:

4.1.1 Review literature and prominent vulnerability databases

This was done as part of the background research and the results can be seen in Section 3.3.2 The work done in this task was to review literature in

- Aspect-Oriented Programming
- Security Vulnerabilities
- Language Design

4.1.2 Identify additional capabilities that would be desirable in a Security Aspect Language

The results of this work as present as part of Section 3.4.3. A large part of the resulting aspect language was due to the work done in analyzing and ascertaining what features were required for addressing security vulnerabilities, while still having a useable language.

4.1.3 Identify, acquire and analyze off-the-shelf tools that have potential to be integrated.

The results from this task indicated that there were no OTS applications available that could be usefully integrated into the AOSF framework. The AOSF framework, however, does rely on the rx package, which is a freely available regular expression library. See Section 3.3.2.

4.1.4 Design the SAL programming language syntax, including functionality for aspect inheritance.

This task was performed as part of the effort in building the AOSF framework, of which the aspect language forms an integral part. More details of the aspect language can be found in Section 3.4.3.

4.1.5 Design a simple tree-based language for use as an intermediate representation for parsed SAL programs.

This task was performed as part of developing the aspect weaver (see Section 3.4.4). The intermediate representation (IR) is the basis for being able to manipulate and transform source code.

4.1.6 Identify and acquire software packages that contain known vulnerabilities that can be used as a basis for testing SAL.

This was done on a continual basis throughout the project. As can be seen in Section 3.6, we did validate the AOSF framework on several open-source applications that contained known vulnerabilities.

4.1.7 Develop Software:

4.1.7.1 Develop a lexical analyzer and parser for the aspect framework

This task was performed as part of the aspect weaver development effort (see Section 3.4.4). It transforms programs written in the C-Saw aspect language to the intermediate tree language developed in task 4.1.5.

4.1.7.2 Develop an Aspect Weaver for C programs

The aspect weaver was developed to take a C program and a C-Saw specification (in its intermediate tree-based form), and convert them to a single "woven" program. The output of the Aspect Weaver is a C program that compiles under the same environments as the original C program. The output is a modified version of the original program such that only the changes necessary and sufficient to satisfy the associated security specification were made. The aspect weaver is the transformation engine behind the AOSF framework and, as such, is one of the three main components of the framework. The aspect weaver is detailed in Section 3.4.4.

4.1.7.3 Develop an Aspect Weaver for PE format (windows NT) binaries.

This task was not performed due to the change of direction at the end of the first year of the effort. The new direction focused on expanding the AOSF framework to work with higher-level security issues and provide aspects to address these, as talked about in Section 3.4.5.2.

4.1.7.4 Apply SAL to several freely available software packages with know vulnerabilities

The AOSF framework was used upon multiple network-oriented applications with known vulnerabilities, in order to validate that the use of the framework would result in the vulnerabilities being corrected. This is shown in Section 3.6 below.

3.6. Experimentation and Validation

The AOSF was validated upon several popular, open-source applications such as wu-ftpd, bind and openSSL. We used versions of applications with known vulnerabilities in order to be able to validate the viability of the AOSF. The experiments were simple in that we used the unmodified versions of the applications and exploited them using the known vulnerabilities as the control set. We then applied appropriate aspects to these applications and tried the same exploits on them. In addition, we also performed regression tests to ensure that we had not changed any of the base functionality of the application. All of the vulnerabilities in these applications were corrected by the use of the AOSF.

The table below presents some of the results from our experiments on the wu-ftpd application, which is a FTP server. In addition to performing the basic functionality and security tests, we also examined the performance costs of applying these aspects. Typically, the cost was not statistically significant. The one exception to this is the encryption aspect, which included the actual overhead to perform the encryption of the data being transferred.

Aspect	**Percentage Increase in post-processed LOC**	**Ratio of join points to cut points**	**Performance penalty**	**Portability of aspect**
Format String	27%	1:2	< 1%	High
Buffer overflow	12%	1:165	< 2%	Medium
TOCTTOU	5%	1:85	<1%	High
Input Sanitization	1.5%	1:1	<1%	Low
Encryption	190%	3:4	~ 600%[1]	Low
Signal Protection	2%	1:57	<1%	High

[1] This is due to the addition of encryption of the data file being transferred, not just the addition of the aspect. This cost would have to be paid in order to encrypt the data using non-AOP means, too.

3.7. References

[1] U. Becker. “A Design-Based Aspect Language for Distribution Control.” *ECOOP '98 Workshop on Aspect-Oriented Programming*, 1998.

[2] L. Bergmans, M. Aksits. “Composing Crosscutting Concerns using Composition Filters.” *Communications of the ACM*, Oct. 2001, Vol. 44, Iss. 10.

[3] K. Bollert. “On Weaving Aspects.” In *International Workshop on Aspect Oriented Programming at ECOOP*, 1999.

[4] S. Clarke et al. “Subject Oriented Design: Towards Improved Alignment of Requirements, Design, and Code.” *Proceedings of the 1999 ACM SIGPLAN conference on OO programming, systems, languages, and applications*. Oct. 1999, vol 34, Iss. 10.

[5] Y. Coady, et al. "Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code" In *Proceedings of Foundations of Software Engineering*. Vienna, Austria, September 2001.

[6] C. Constantinedes, Atef Bader, T. Elrad, P. Netinant, M. Fayad. “Designing an Aspect-Oriented Framework in an Object-Oriented Environment.” *ACM Computing Surveys (CSUR)*. March 2000.

[7] C. Cowan et al. "StackGuard 1.1: Stack Smashing Protection for Shared Libraries." In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1998.

[8] Crispin Cowan et al. "FormatGuard: Automatic Protection From printf Format String Vulnerabilities", in the *10th USENIX Security Symposium*, Washington, DC, August 2001.

[9] R. DeLine and M. Fähndrich. "Enforcing high-level protocols in low-level software." In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2001, pages 59-69.

[10] T. Elrad, R. Filman, Atef Bader. “Aspect Oriented Programming.” *Communications of the ACM*, Oct. 2001. Vol. 44, Iss. 10.

[11] P. Fradet and M. Sudhold. “AOP: Towards a Generic Framework Using Program Transformation and Analysis.” In *AOP98*. 1998.

[12] P. Fradet and M. Sudholt. “An Aspect Language for Robust Programming.” In *International Workshop on Aspect Oriented Programming at ECOOP*. 1999.

[13] ITS4. Cigital, Inc. <http://www.cigital.com/its4>

[14] T. Jim et al. “Cyclone: A Safe Dialect of C” *USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[15] G. Kiczales, et al. “Aspect Oriented Programming”, *ACM Computing Surveys. Dec*. 1996.

[16] G. Kiczales et al. “Getting Started with AspectJ.” *Communications of the ACM* Oct 2001, Vol. 44, Iss. 10.

[17] H. Kim. *AspectC# Website* Trinity College, Dublin. <http://www.cs.tcd.ie/Howard.Kim/aspectcsharp/>

[18] K. Lieberherr, D. Orleans, J. Ovlinger. “Aspect-Oriented Programming with Adaptive Methods.” *Communications of the ACM* Oct 2001, Vol. 44, Iss. 10.

[19] G. Murphy, et al. “Does Aspect-Oriented Programming Work?” *Communications of the ACM* Oct 2001, Vol. 44, Iss. 10.

[20] H. Ossher, P. Tarr. “Using Multidimensional Separation of Concerns to (Re)shape Evolving Software. *Communications of the ACM* Oct 2001, Vol. 44, Iss. 10.

[21] O Spinczyk, A. Gal and W. Schröder-Preikschat, "AspectC++: An Aspect-Oriented Extension to C++", *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 18-21, 2002

[22] P. Tarr, et al. “N Degrees of Separation: Multi-Dimensional Separation of Concerns.” *Proc. of the 1999 Int'l. Conference on Software Engineering*, May 1999.

[23] E. Truyen, B. N. Jorgensen, W. Joosen, P. Verbaeten. “Aspects for Run-Time Component Integration.” *International Workshop on Aspects and Dimensional Computing at ECOOP*, 2000.

[24] Viega et al. "RATS, the Rough Auditing Tool for Security." http://www.securesoftware.com/rats.php.

[25] J. Viega and G. McGraw. *Building Secure Software*. Boston: Addison-Wesley, September 2001.

[26] R. Walker, E. Baniassad, G. Murphy. “An Initial Assessment of AOP.” *Proc. of the 1999 Int'l Conference on Software Engineering*, 1999.

[27] M. Weber, V. Shah and C. Ren. “A Case Study in Detecting Software Security Vulnerabilities Using Constraint Optimization” *IEEE Workshop on Source Code Analysis and Manipulation*, November 2001, Florence, Italy.

APPENDIX A: Security Aspect Language (C-Saw) BNF

This appendix lists the current BNF grammar for the languages used by the prototype system i.e. the grammar for the C-Saw (Security Aspect Language), which of necessity includes the grammar for the C language since the C-Saw is based on it.

```
/*
 * handles the start of parsing. the first token passed back by the
lexer
 * determines what kind of file the parser is about to try to parse.
 */
Start
  : CCODE c_file
  | CSAWCODE CSAW_Aspect_Decl

/*
 * handles an aspect. this is the only valid top level entry in an
 * aspect file. only one may be present per file, but the change to
 * supporting multiple aspects per file is trivial.
 */
CSAW_Aspect_Decl
  : ASPECT id CSAW_Aspect_Code

/*
 * handles tha actual content of the apsect declaration. this is
 * essentially a c_file and a location declaration list, but the
 * c_file may be omitted.
 */
CSAW_Aspect_Code
  : '{' '}'
  | '{' CSAW_Location_Decl_List '}'
  | '{' c_file CSAW_Location_Decl_List '}'

/*
 * handles a list of location declarations.
 */
CSAW_Location_Decl_List
  : CSAW_Location_Decl
  | CSAW_Location_Decl_List CSAW_Location_Decl

/*
 * handles a single location declaration. this is basically a
 * location specification followed by the code that holds the
 * locations's code quantifiers.
 */
CSAW_Location_Decl
  : CSAW_Location CSAW_Location_Code

/*
 * handles a single location specification. this consists of either
 * a point qualifier, or a point qualifier and an area qualifier
 * expression.
 */
CSAW_Location
  : CSAW_Point_Qual
  | CSAW_Point_Qual ':' CSAW_Area_Exp
```

```
/*
 * handles an area expression that is a binary logical-or expression.
 */
CSAW_Area_Exp
  : CSAW_Area_Term
  | CSAW_Area_Exp OR_OP CSAW_Area_Term

/*
 * handles an area expression that is a binary logical-and expression.
 */
CSAW_Area_Term
  : CSAW_Area_Term AND_OP CSAW_Area_Factor
  | CSAW_Area_Factor

/*
 * handles an area expression that is a unary logical-not expression.
 */
CSAW_Area_Factor
  : '!' CSAW_Area_Factor
  | CSAW_Area_Unit

/*
 * handles an expression's smallest unit. this can consist of a more
 * complicated expression that is parenthesized.
 */
CSAW_Area_Unit
  : '(' CSAW_Area_Exp ')'
  | CSAW_Area_Qual

/*
 * handles a single point qualifier.
 */
CSAW_Point_Qual
  : CSAW_PQ_FuncDecl
  | CSAW_PQ_FuncDef
  | CSAW_PQ_FuncCall
  | CSAW_PQ_LabMatch
  | CSAW_PQ_Lab_Range

/*
 * handles the function declaration point qualifier.
 */
CSAW_PQ_FuncDecl
  : FUNCDECL '<' CSAW_Signature '>'

/*
 * handles the function definition point qualifier.
 */
CSAW_PQ_FuncDef
  : FUNCDEF '<' CSAW_Signature '>'

/*
 * handles the function call point qualifier.
 */
CSAW_PQ_FuncCall
  : FUNCCALL '<' CSAW_Signature '>'
```

```
/*
 * handles the label match point qualifier.
 */
CSAW_PQ_LabMatch
  : LABELMATCH '<' CSAW_Name_Pat '>'

/*
 * handles the label range point qualifier.
 */
CSAW_PQ_Lab_Range
  : LABELRANGE '<' id ',' id '>'

/*
 * handle a name pattern. this is used when a name can either be
 * wild of not.
 */
CSAW_Name_Pat
  : wild_id
  | id

/*
 * handles a single area qualifier.
 */
CSAW_Area_Qual
  : CSAW_AQ_InModule
  | CSAW_AQ_InFunc
  | CSAW_AQ_InAspect

/*
 * handles the in module area qualifier.
 */
CSAW_AQ_InModule
  : INMODULE '<' CSAW_Name_Pat '>'

/*
 * handles the in function area qualifier.
 */
CSAW_AQ_InFunc
  : INFUNC '<' CSAW_Signature '>'

/*
 * handles the in aspect area qualifier.
 */
CSAW_AQ_InAspect
  : INASPECT '<' CSAW_Name_Pat '>'

/*
 * handles the code that appears in a location declaration. this can be
 * just location contents, or a declaration list followed by location
 * contents.
 */
CSAW_Location_Code
  : '{' '}'
  | '{' CSAW_Location_Contents '}'
  | '{' declaration_list CSAW_Location_Contents '}'

/*
```

```
 * handles location contents. this is a list of location content's.
 */
CSAW_Location_Contents
  : CSAW_Location_Content
  | CSAW_Location_Contents CSAW_Location_Content

/*
 * handles a single location content (a code quantifier).
 */
CSAW_Location_Content
  : CSAW_CQ_Before_Block
  | CSAW_CQ_After_Block
  | CSAW_CQ_Replace_Block

/*
 * handles a before code quantifier.
 */
CSAW_CQ_Before_Block
  : BEFORE compound_statement

/*
 * handles an after code quantifier.
 */
CSAW_CQ_After_Block
  : AFTER compound_statement

/*
 * handles a replace code quantifier.
 */
CSAW_CQ_Replace_Block
  : REPLACE compound_statement

/*
 * handles a signature. this is similar to what appears in a single-
 * declarator declaration except the lexer will parse these in
 * C_WILD_MODE (i.e. accept type and name wildcards).
 */
CSAW_Signature
  : typed_declaration_specifier_list declarator
  | declaration_modifiers notype_declarator

/*
 * handles a simple double quoted string literal
 */
string
  : STRING_LITERAL
  | string STRING_LITERAL

/*
 * handles an expression's smallest unit. this also includes
 * a more complicated expression that has been parenthesized.
 */
primary_expr
  : id
  | '(' compound_statement ')' /* gcc only */
  | CONSTANT
  | FLOATCONSTANT
```

```
  | string
  | '(' expr ')'

/*
 * handles an expression that has postfix operations associated with
 * it. this includes function calls, array indexes, member access, and
 * increment/decrement operations.
 */
postfix_expr
  : primary_expr
  | postfix_expr '[' expr ']'
  | postfix_expr '(' ')'
  | postfix_expr '(' argument_expr_list ')'
  | postfix_expr '.' identifier
  | postfix_expr PTR_OP identifier
  | postfix_expr INC_OP
  | postfix_expr DEC_OP

/*
 * handles a comma delimited set of expressions which is used primarily
 * for making expressions that are function calls.
 */
argument_expr_list
  : assignment_expr
  | argument_expr_list ',' assignment_expr

/*
 * handles any expressions with associated prefix unary expressions.
 * this includes increment/decrement, unary operations, or sizeof
 * operations.
 */
unary_expr
  : postfix_expr
  | INC_OP unary_expr
  | DEC_OP unary_expr
  | unary_operator cast_expr
  | SIZEOF unary_expr
  | SIZEOF '(' type_name ')'

/*
 * handles unary operators themselves, including unary plus/minus,
 * bitwise/logical not, and reference/dereference.
 */
unary_operator
  : '&'
  | '*'
  | '+'
  | '-'
  | '~'
  | '!'

/*
 * handles an expressions with cast information preceding. the cast
 * information is a type_name.
 */
cast_expr
  : unary_expr
```

```
  | '(' type_name ')' cast_expr
  | '(' type_name ')' '{' initializer_list '}' /*gcc only */
  | '(' type_name ')' '{' initializer_list ',' '}' /* gcc only */

/*
 * handles binary expressions with multiplicative precedence.
 */
multiplicative_expr
  : cast_expr
  | multiplicative_expr '*' cast_expr
  | multiplicative_expr '/' cast_expr
  | multiplicative_expr '%' cast_expr

/*
 * handles binary expressions with additive precedence.
 */
additive_expr
  : multiplicative_expr
  | additive_expr '+' multiplicative_expr
  | additive_expr '-' multiplicative_expr

/*
 * handles binary expressions with bit-shift precedence.
 */
shift_expr
  : additive_expr
  | shift_expr LEFT_OP additive_expr
  | shift_expr RIGHT_OP additive_expr

/*
 * handles binary expressions with relational precedence.
 */
relational_expr
  : shift_expr
  | relational_expr '<' shift_expr
  | relational_expr '>' shift_expr
  | relational_expr LE_OP shift_expr
  | relational_expr GE_OP shift_expr

/*
 * handles binary expressions with equality precedence.
 */
equality_expr
  : relational_expr
  | equality_expr EQ_OP relational_expr
  | equality_expr NE_OP relational_expr

/*
 * handles binary expressions with bitwise-and precedence.
 */
and_expr
  : equality_expr
  | and_expr '&' equality_expr

/*
 * handles binary expressions with bitwise-xor precedence.
 */
```

```
exclusive_or_expr
  : and_expr
  | exclusive_or_expr '^' and_expr

/*
 * handles binary expressions with bitwise-or precedence.
 */
inclusive_or_expr
  : exclusive_or_expr
  | inclusive_or_expr '|' exclusive_or_expr

/*
 * handles binary expressions with logical-and precedence.
 */
logical_and_expr
  : inclusive_or_expr
  | logical_and_expr AND_OP inclusive_or_expr

/*
 * handles binary expressions with logical-or precedence.
 */
logical_or_expr
  : logical_and_expr
  | logical_or_expr OR_OP logical_and_expr

/*
 * handles ternary expressions with logical-and precedence.
 */
conditional_expr
  : logical_or_expr
  | logical_or_expr '?' logical_or_expr ':' conditional_expr

/*
 * handles binary expressions with assignment precedence.
 */
assignment_expr
  : conditional_expr
  | unary_expr assignment_operator assignment_expr

/*
 * handles the assignment operators. this includes the standard
 * assignment operator and other operation-assignment operators.
 */
assignment_operator
  : '='
  | MUL_ASSIGN
  | DIV_ASSIGN
  | MOD_ASSIGN
  | ADD_ASSIGN
  | SUB_ASSIGN
  | LEFT_ASSIGN
  | RIGHT_ASSIGN
  | AND_ASSIGN
  | XOR_ASSIGN
  | OR_ASSIGN

/*
```

```
 * handles any expression or comma delimited list of expressions.
 * this starts the normal sequence of rules that allows for proper
 * parsing with precedence rules.
 */
expr
  : assignment_expr
  | expr ',' assignment_expr

/*
 * handles any expressions that can be evaluated to constant value
 * during compile time. this is not always true since parenthesized
 * expressions break this, but it does succeed in assuring no top
 * precedence level assignments are allowed.
 */
constant_expr
  : conditional_expr

/*
 * handles the definition of a function. this keeps
 * the proper rules in mind for handle typedef'd names.
 */
function_definition
  : notype_declarator function_body
  | declaration_modifiers notype_declarator function_body
  | typed_declaration_specifier_list declarator function_body

/*
 * handles the declaration of a variable or function. this keeps
 * the proper rules in mind for handle typedef'd names.
 */
declaration
  : declaration_modifiers notype_init_declarator_list ';'
  | typed_declaration_specifier_list init_declarator_list ';'
  | declaration_modifiers ';'
  | typed_declaration_specifier_list ';'

/*
 * handles a list of declarators where each could possibly have
 * an associated initializer. the declarators in this list must
 * be notype_init_declarator's.
 */
notype_init_declarator_list
  : notype_init_declarator
  | notype_init_declarator_list ',' notype_init_declarator

/*
 * handles a list of declarators where each could possibly have
 * an associated initializer. the declarators in this list must
 * be init_declarator's.
 */
init_declarator_list
  : init_declarator
  | init_declarator_list ',' init_declarator

/*
 * handles declarators, some possibly with initializers. the
 * declarators used here can be either aftertype_declarator's
```

```
 * or notype_declarator's (see the declarator rule).
 */
init_declarator
  : declarator
  | declarator '=' initializer

/*
 * handles declarators, some possibly with initializers. the
 * declarators used here must be notype_declarator's.
 */
notype_init_declarator
  : notype_declarator
  | notype_declarator '=' initializer

/*
 * handles a declarator. this allows the declarator to be either
 * an aftertype_declarator or a notype_declarator. see the rules
 * for each to see the difference.
 */
declarator
  : aftertype_declarator
  | notype_declarator

/*
 * handles the aftertype type of declarator. this indicates that a
 * pointer may be used in combination with the declarator's predicate.
 * this type of declarator that must use a previously declared
 * typedef'd name as the identifier of the declarator.
 */
aftertype_declarator
  : aftertype_declarator2
  | pointer aftertype_declarator2

/*
 * handles the predicate of an aftertype_declarator. this includes
 * constructs that indicate arrays, functions, a previously declared
 * typedef'd name, or grouping of an aftertype_declarator for
 * precedence reasons.
 */
aftertype_declarator2
  : '(' aftertype_declarator ')'
  | aftertype_declarator2 '[' ']'
  | aftertype_declarator2 '[' constant_expr ']'
  | aftertype_declarator2 '(' ')'
  | aftertype_declarator2 '(' parameter_type_list ')'
  | aftertype_declarator2 '(' parameter_identifier_list ')'
  | type

/*
 * handles the notype type of declarator. this indicates that a pointer
 * may be used in combination with the declarator's predicate.
 * this type of declarator that must not use a previously declared
 * typedef'd name as the identifier of the declarator. this rule
 * also allows for wild_id's to be used, but this will only occur
 * when the lexer is in the right mode (i.e. in particular places
 * in C-Saw and never in C).
 */
```

```
notype_declarator
  : notype_declarator2
  | pointer notype_declarator2

/*
 * handles the predicate of an notype_declarator. this includes
 * constructs that indicate arrays, functions, identifiers, or grouping
 * of a notype_declarator for precedence reasons.
 */
notype_declarator2
  : '(' notype_declarator ')'
  | notype_declarator2 '[' ']'
  | notype_declarator2 '[' constant_expr ']'
  | notype_declarator2 '(' ')'
  | notype_declarator2 '(' parameter_type_list ')'
  | notype_declarator2 '(' parameter_identifier_list ')'
  | id
  | wild_id

/*
 * handles a list of declaration specifiers that do not provide any
 * concrete type information.
 */
declaration_modifiers
  : type_qualifier
  | storage_class_specifier
  | declaration_modifiers type_qualifier
  | declaration_modifiers storage_class_specifier

/*
 * handles a list of declaration specifiers that do provide concrete
 * type information.
 */
typed_declaration_specifier_list
  : declaration_modifiers type_specifier res_declaration_specifiers
  | type_specifier res_declaration_specifiers

/*
 * handles any single construct that provides concrete type
information.
 * this includes standard keywords, structure specifiers, enumeration
 * specifiers, previously declared typedef'd names, and typeof
 * constructs. while the "any" keyword is valid here, this will only
 * occur when the lexer is in the right mode (i.e. in particular places
 * in C-Saw and never in C).
 */
type_specifier
  : ANY
  | CHAR
  | INT
  | FLOAT
  | DOUBLE
  | VOID
  | SHORT
  | LONG
  | SIGNED
  | UNSIGNED
```

```
  | struct_or_union_specifier
  | enum_specifier
  | type
  | TYPEOF '(' type_name ')' /* gcc only */
  | TYPEOF '(' expr ')' /* gcc only */

/*
 * handles a list (possibly empty) of declaration specifiers that
 * can include any type of specifier (see
 * res_declaration_specifier rule).
 */
res_declaration_specifiers
  : /*empty*/
  | res_declaration_specifiers res_declaration_specifier

/*
 * handles any single declaration specifier, regardless of whether or
 * not it provides concrete type information or is a storage class
 * specifier. see the exception for the "any" keyword mentioned above.
 */
res_declaration_specifier
  : storage_class_specifier
  | ANY
  | CHAR
  | INT
  | FLOAT
  | DOUBLE
  | VOID
  | SHORT
  | LONG
  | SIGNED
  | UNSIGNED
  | struct_or_union_specifier
  | enum_specifier
  | type_qualifier

/*
 * handles any single declaration specifier that is used solely to
 * qualify but not specify a type.
 */
type_qualifier
  : CONST
  | VOLATILE

/*
 * handles a list of declaration specifiers or which each is a type
 * qualifier.
 */
type_qualifier_list
  : type_qualifier
  | type_qualifier_list type_qualifier

/*
 * handles a list of declaration specifiers that do provide concrete
 * type information without allowing any storage class specifiers.
 */
typed_type_specifiers
```

```
  : type_specifier res_type_spec_qualifiers
  | type_qualifier_list type_specifier res_type_spec_qualifiers

/*
 * handles a list (possibly empty) of declaration specifiers that
 * can include any type of specifier except storage class specifiers
 * (see res_type_spec_qualifier rule).
 */
res_type_spec_qualifiers
  : /*empty*/
  | res_type_spec_qualifiers res_type_spec_qualifier

/*
 * handles a single declaration specifier. it can be either a concrete
 * type specifier or not, but regardless, it may not be a storage class
 * specifier. see the exception for the "any" keyword mentioned above.
 */
res_type_spec_qualifier
  : ANY
  | CHAR
  | INT
  | FLOAT
  | DOUBLE
  | VOID
  | SHORT
  | LONG
  | SIGNED
  | UNSIGNED
  | struct_or_union_specifier
  | enum_specifier
  | type_qualifier

/*
 * handles a declaration specifier that is a keyword to indicate that
 * the type specified with this declaration specifier is of a
 * particular standard C storage class.
 */
storage_class_specifier
  : TYPEDEF
  | EXTERN
  | INLINE /* gcc only */
  | STATIC
  | AUTO
  | REGISTER

/*
 * handles a declaration specifier that is a structure. this specifier
 * can either declare a structure, define a structure, or refer to a
 * previously declared/defined tagged structure.
 */
struct_or_union_specifier
  : struct_or_union identifier '{' struct_declaration_list '}'
  | struct_or_union '{' struct_declaration_list '}'
  | struct_or_union identifier

/*
 * handles the keyword that indicates either a union of a structure.
```

```
 */
struct_or_union
  : STRUCT
  | UNION

/*
 * handles a list (possibly empty and/or possibly with multiple
 * back-to-back semicolons) of declarations for structure members.
 */
struct_declaration_list
  : /*empty*/
  | struct_declaration_list struct_declaration ';'
  | struct_declaration_list ';' /* gcc only */

/*
 * handles a single declaration in a structure. this keeps in mind the
 * possible presence of typedef'd names used in declarators. a
typedef'd
 * name shadowed here is only shadowed for declarations proceding the
 * shadowing declaration in the containing list.
 */
struct_declaration
  : typed_type_specifiers struct_declarator_list
  | typed_type_specifiers /* gcc only */
  | type_qualifier_list notype_struct_declarator_list
  | type_qualifier_list /* gcc only */

/*
 * handles a list of declarators for structure members. the declarators
 * in this list may or may not use typedef'd names.
 */
struct_declarator_list
  : struct_declarator
  | struct_declarator_list ',' struct_declarator

/*
 * handles a single declarator for a structure member. this also allows
 * for bitfield expressions. this type of declarator may or may not
 * contain typedef'd names.
 */
struct_declarator
  : declarator
  | ':' constant_expr
  | declarator ':' constant_expr

/*
 * handles a list of declarators for structure members. the declarators
 * in this list may not use typedef'd names.
 */
notype_struct_declarator_list
  : notype_struct_declarator
  | notype_struct_declarator_list ',' notype_struct_declarator

/*
 * handles a single declarator for a structure member. this also allows
 * for bitfield expressions. this type of declarator may not contain
 * typedef'd names.
```

```
 */
notype_struct_declarator
  : notype_declarator
  | ':' constant_expr
  | notype_declarator ':' constant_expr

/*
 * handles a declaration specifier that is an enumeration. this is used
 * to declare, define, or refer to a previously declared/defined tagged
 * enumeration.
 */
enum_specifier
  : ENUM '{' enumerator_list '}'
  | ENUM '{' enumerator_list ',' '}'
  | ENUM identifier '{' enumerator_list '}'
  | ENUM identifier '{' enumerator_list ',' '}'
  | ENUM identifier

/*
 * handles the list of items in an enumeration.
 */
enumerator_list
  : enumerator
  | enumerator_list ',' enumerator

/*
 * handles a single item in an enumeration. this is either an
identifier
 * or an identifier,constant_expr pair.
 */
enumerator
  : identifier
  | identifier '=' constant_expr

/*
 * handles a pointer, which may be a list of pointers. each pointer in
 * the list may have declaration specifiers that are type qualifiers.
 */
pointer
  : '*'
  | '*' type_qualifier_list
  | '*' pointer
  | '*' type_qualifier_list pointer

/*
 * handles a parameter list that is given by identifer only. this is
 * primarily used in K&R type function declarations/definitions. this
 * also allows for varargs functions.
 */
parameter_identifier_list
  : identifier_list
  | identifier_list ',' ELIPSIS

/*
 * handles a list of identifiers. this is only used by the
 * parameter_identifier_list rule.
 */
```

```
identifier_list
  : id
  | identifier_list ',' id

/*
 * handles a list of parameter declarations that may have varargs.
 */
parameter_type_list
  : parameter_list
  | parameter_list ',' ELIPSIS

/*
 * handles a list of parameter declarations.
 */
parameter_list
  : parameter_declaration
  | parameter_list ',' parameter_declaration

/*
 * handles a parameter declaration that is used in the declaration
 * of a function's arguments. this allows for both named declarators
 * and abstract declarators (those without names). additionally it
 * allows for any combination of declaration specifiers and declarators
 * with or without typedef'd names.
 */
parameter_declaration
  : typed_declaration_specifier_list parameter_declarator
  | typed_declaration_specifier_list notype_declarator
  | typed_declaration_specifier_list abstract_declarator
  | typed_declaration_specifier_list
  | declaration_modifiers notype_declarator
  | declaration_modifiers abstract_declarator
  | declaration_modifiers

/*
 * handles a parameter declarator (which is really only one of several
 * types of declarator types allowed in parameter declarations).
 * this is essentially the same as an aftertype_declarator except in
 * the allowable declarator predicates.
 */
parameter_declarator
  : parameter_declarator2
  | pointer parameter_declarator2

/*
 * handles a parameter declarator's predicate. this is identical to
 * aftertype_declarator's viable predicate list except for the grouping
 * rule which has been removed from this one.
 */
parameter_declarator2
  : parameter_declarator2 '[' ']'
  | parameter_declarator2 '[' constant_expr ']'
  | parameter_declarator2 '(' ')'
  | parameter_declarator2 '(' parameter_type_list ')'
  | parameter_declarator2 '(' parameter_identifier_list ')'
  | type
```

```
/*
 * handles a type name construct. this is essentially only used by
 * the sizeof expressions. this allows only for abstract declarators
 * and no concrete declarators (those with identifiers/typedef'd names)
 * can be used.
 */
type_name
  : typed_type_specifiers
  | typed_type_specifiers abstract_declarator
  | type_qualifier_list
  | type_qualifier_list abstract_declarator

/*
 * handles a declarator that is abstract, i.e. it will contain no name.
 * a name is simply the identifier in a notype_declarator or the type
 * in an aftertype_declarator.
 */
abstract_declarator
  : pointer
  | abstract_declarator2
  | pointer abstract_declarator2

/*
 * handles all the possible predicates for an abstract declarator. the
 * rule for a name was replaced with several rules that terminate the
 * predicate with one of the possible endings.
 */
abstract_declarator2
  : '(' abstract_declarator ')'
  | '[' ']'
  | '[' constant_expr ']'
  | abstract_declarator2 '[' ']'
  | abstract_declarator2 '[' constant_expr ']'
  | '(' ')'
  | '(' parameter_type_list ')'
  | abstract_declarator2 '(' ')'
  | abstract_declarator2 '(' parameter_type_list ')'

/*
 * handles an expression or a braced set that indicates an initializer
 * to a declaration.
 */
initializer
  : assignment_expr
  | '{' initializer_list '}'
  | '{' initializer_list ',' '}'

/*
 * handles possible types of initializer's allowed in a braced set.
 * this rule is only needed to handle gcc and C9X extensions.
 */
initializer2
  : designator_list '=' initializer /* gcc only */
  | designator initializer /* gcc only */
  | identifier ':' initializer /* C9X only */
  | initializer
```

```
/*
 * handles handles a list of initializers. this really handles a list
 * of initializer2's but this is only because this rule is for
 * braced set contents and the initializer2 rules must be allowed.
 * without the extensions in the initializer2 rule, this could simply
 * be a list of initializer's.
 */
initializer_list
  : initializer2
  | initializer_list ',' initializer2

/*
 * handles a list of designators. these are gcc extension for different
 * conventions on braced set initializers.
 */
designator_list
  : designator
  | designator_list designator

/*
 * handles a single designator. these represent three different gcc
only
 * ways of making initializers.
 */
designator
  : '.' identifier
  | '[' assignment_expr ']'
  | '[' assignment_expr ELIPSIS assignment_expr ']'

/*
 * handles any type of statement.
 */
statement
  : labeled_statement
  | compound_statement
  | expression_statement
  | selection_statement
  | iteration_statement
  | jump_statement

/*
 * handles a labeled statement, a case statement, or a default
 * statement.
 */
labeled_statement
  : identifier ':' statement
  | CASE constant_expr ':' statement
  | CASE constant_expr ELIPSIS constant_expr ':' statement
  | DEFAULT ':' statement

/*
 * handles a compound statement. this may be composed of a declaration
 * list and a statement list (either or both can be omitted).
 */
compound_statement
  : '{' '}'
  | '{' statement_list '}'
```

```
  | '{' declaration_list '}'
  | '{' declaration_list statement_list '}'

/*
 * handles a list of declarations
 */
declaration_list
  : declaration
  | declaration_list declaration

/*
 * handles a list of statements
 */
statement_list
  : statement
  | statement_list statement

/*
 * handles a statement that is an expression.
 */
expression_statement
  : ';'
  | expr ';'

/*
 * handles a statement that is an if statement, an if-else statement,
 * or a switch statement.
 */
selection_statement
  : IF '(' expr ')' statement
  | IF '(' expr ')' statement ELSE statement
  | SWITCH '(' expr ')' statement

/*
 * handles a statement that is a while statement, a do-while statement,
 * or a for statement.
 */
iteration_statement
  : WHILE '(' expr ')' statement
  | DO statement WHILE '(' expr ')' ';'
  | FOR '(' ';' ';' ')' statement
  | FOR '(' ';' ';' expr ')' statement
  | FOR '(' ';' expr ';' ')' statement
  | FOR '(' ';' expr ';' expr ')' statement
  | FOR '(' expr ';' ';' ')' statement
  | FOR '(' expr ';' ';' expr ')' statement
  | FOR '(' expr ';' expr ';' ')' statement
  | FOR '(' expr ';' expr ';' expr ')' statement

/*
 * handles a statement that is a goto statement, a continue statement,
 * a break statement, or a return statement.
 */
jump_statement
  : GOTO identifier ';'
  | CONTINUE ';'
  | BREAK ';'
```

```
  | RETURN ';'
  | RETURN expr ';'

/*
 * handles a list of external definitions that compose the contents
 * of a C file.
 */
c_file
  : external_definition
  | c_file external_definition

/*
 * handles an external definition which can be either a declaration or
 * a function definition.
 */
external_definition
  : function_definition
  | declaration

/*
 * handles the body of a function. this is normally just a compound
 * statement except in the case of K&R style function definition.
 */
function_body
  : compound_statement
  | declaration_list compound_statement

/*
 * handles an identifier or a previously declared typedef'd name.
 */
identifier
  : id
  | type

/*
 * handles a previously declared typedef'd name.
 */
type
  : TYPE_NAME

/*
 * handles an identifier. this also allows for an escaped identifier
 * or an escaped previously declared typedef'd name. the token that is
 * used to indicate scoping can only be returned in special lexer modes
 * (i.e. the mode is present when parsing certain parts of a C-Saw
file,
 * and never while parsing C files).
 */
id
  : IDENTIFIER
  | SCOPE_OP IDENTIFIER
  | SCOPE_OP TYPE_NAME

/*
 * handles a wild identifier, i.e. one that is composed of letters,
 * numbers, and the wildcard character.
 */
```

```
wild_id
  : WILD_IDENTIFIER
\end{verbatim}
\normalsize
```